Of course. As a senior software engineer, my primary goal is to translate a product vision into a clear, actionable, and robust technical blueprint. This document will serve as that blueprint for the development team, ensuring every feature is built to specification, is scalable, and aligns with the overall architecture.

Here is the detailed feature specification document for the School Kitchen Inventory Management application.

**File System**

A structured file system is crucial for maintainability, especially when targeting multiple platforms. We will separate the shared logic from the platform-specific UI code.

* **Root Project Folder:** KitchenInventoryApp/
  + src/
    - Core/ (This will be a .NET Standard Class Library project)
      * Data/
        + KitchenDbContext.cs (Entity Framework Core or Dapper context for SQLite)
        + Repositories/ (Interfaces and implementations for data access)

IItemRepository.cs

ItemRepository.cs

IUserRepository.cs

UserRepository.cs

ICategoryRepository.cs

CategoryRepository.cs

IHistoryRepository.cs

HistoryRepository.cs

* + - * + Models/ (Plain C# objects representing database entities)

Item.cs

User.cs

Category.cs

UsageHistory.cs

* + - * + Migrations/ (Handled by Entity Framework Core for schema versioning)
      * BusinessLogic/
        + Services/

AuthenticationService.cs

InventoryService.cs

ReportingService.cs

SuggestionService.cs

* + - * + ViewModels/ (Shared ViewModel base classes, if using MVVM)
      * Constants/
        + Roles.cs (e.g., public const string Admin = "Admin";)
    - WindowsUI/ (This will be a C# WPF Project)
      * Views/ (XAML files for windows and user controls)
        + LoginWindow.xaml
        + MainWindow.xaml
        + InventoryView.xaml
        + UserDetailsView.xaml
        + ReportsView.xaml
      * ViewModels/ (WPF-specific ViewModels that wrap the Core logic)
        + LoginViewModel.cs
        + MainViewModel.cs
        + InventoryViewModel.cs
      * Assets/ (Images, icons, fonts)
      * App.xaml.cs (Application entry point)
    - MacUI/ (This will be a SwiftUI Project)
      * Views/ (SwiftUI views)
        + LoginView.swift
        + MainView.swift
        + InventoryListView.swift
      * ViewModels/ (SwiftUI-specific ViewModels/State Objects)
        + AuthViewModel.swift
        + InventoryViewModel.swift
      * Assets.xcassets
  + docs/
    - Architecture.md
    - DatabaseSchema.md
  + tests/
    - Core.Tests/ (Unit tests for the business logic)
    - WindowsUI.Tests/ (UI-specific tests)

**Feature 1: Item & Category Management**

* **Feature Goal:** Provide a comprehensive inventory tracking system where users can perform full CRUD (Create, Read, Update, Delete) operations on inventory items and their associated categories.
* **API Relationships:** This feature interacts exclusively with the local SQLite database via the Core library's repository layer. No external APIs are involved.
* **Detailed Feature Requirements:**
  + Full CRUD operations for inventory items (Name, Quantity, Expiration Date, Category).
  + Full CRUD operations for categories (Name), restricted to Admin users.
  + Search and filter functionality on the inventory list based on item name and category.
  + Sort functionality on the inventory list based on name, quantity, or expiration date.
  + Visual alerts for items that are low in stock or nearing their expiration date.
* **Detailed Implementation Guide:**
  + **Database:** Use SQLite managed via Entity Framework Core within the .NET Standard library for cross-platform compatibility.
  + **Entities:**
    - Item: Id (UUID, Primary Key), Name (Text, NOT NULL), Quantity (Integer, NOT NULL), ExpirationDate (Date), CategoryId (UUID, Foreign Key), LowStockThreshold (Integer), IsDeleted (Boolean, default FALSE), CreatedAt (DateTime), UpdatedAt (DateTime).
    - Category: Id (UUID, Primary Key), Name (Text, NOT NULL, UNIQUE), IsDeleted (Boolean, default FALSE), CreatedAt (DateTime), UpdatedAt (DateTime).
  + **CRUD Validation:**
    - **Create Item:** Name cannot be empty. Quantity must be a non-negative integer. Category must be selected.
    - **Create Category:** Name cannot be empty and must be unique.
    - **Read:** The inventory view will fetch all non-deleted items. Search will be performed via a LIKE query on the Name field. Filtering will be an exact match on CategoryId.
    - **Update:** Allow partial updates. When quantity is updated, a record should be added to the UsageHistory table.
    - **Delete:** Use a soft-delete pattern by setting the IsDeleted flag to TRUE. This preserves historical data for reporting and prevents data integrity issues. The UI should hide items where IsDeleted is TRUE.
  + **UI Pattern:** Strictly follow the MVVM (Model-View-ViewModel) pattern. The View (WPF/SwiftUI) will be dumb, only binding to properties and commands on the ViewModel. The ViewModel will contain the presentation logic and will call services in the Core library to perform business operations.
  + **Responsiveness:** The UI must be built to be responsive, especially the main inventory grid, to ensure it is usable on various screen sizes and resolutions common in older hardware. Ensure large, clear fonts and high-contrast color schemes for accessibility.

**Feature 2: User Roles & Access Management**

* **Feature Goal:** Implement a secure authentication and role-based authorization system to control access to different application features based on user responsibilities (Admin, Manager, Staff).
* **API Relationships:** This feature links the authenticated user's role to all other feature operations, acting as a gatekeeper. It interacts only with the local Users table in the SQLite database.
* **Detailed Feature Requirements:**
  + A secure login screen for user authentication.
  + Three distinct user roles with specific permissions:
    - **Admin:** Full CRUD on everything (Items, Categories, Users, Reports).
    - **Manager:** Read access to all data, including reports. Can update item quantities. Cannot edit users or categories.
    - **Staff:** Read access to inventory list. Can only update item quantities. Cannot view reports or access any settings.
  + An Admin-only interface for creating, editing, and deleting user accounts.
* **Detailed Implementation Guide:**
  + **Database Entity:**
    - User: Id (UUID, Primary Key), Username (Text, NOT NULL, UNIQUE), PasswordHash (Text, NOT NULL), Role (Text, NOT NULL), IsDeleted (Boolean, default FALSE), CreatedAt (DateTime), UpdatedAt (DateTime).
  + **Security:**
    - Passwords must **never** be stored in plain text. Use a strong, standard hashing algorithm like **Argon2** or **bcrypt** to generate the PasswordHash. The .NET System.Security.Cryptography libraries can be used for this.
    - The AuthenticationService in the Core library will handle login verification by hashing the user-provided password and comparing it to the stored PasswordHash.
  + **Authorization:**
    - Upon successful login, the user's Id and Role will be stored in a static session object or application-level state manager.
    - Every action (e.g., opening a view, clicking a button) that requires specific permissions must first check the current user's role against the required role.
    - UI elements for restricted actions (e.g., "Manage Users" button) should be hidden or disabled for unauthorized roles, not just protected on the backend.
  + **User Management UI:** The user management view will be accessible only to Admins. It will allow them to set the Username, Password (on create), and Role for other users. Admins cannot delete their own account or change their own role to a lower-privileged one.

**Feature 3: Rule-Based Reorder Suggestions**

* **Feature Goal:** Provide actionable, automated reorder suggestions to prevent stockouts, based on simple, configurable rules and historical usage data.
* **API Relationships:** This feature relies on data from the Items and UsageHistory tables within the local database. No external AI services are needed for the MVP.
* **Detailed Feature Requirements:**
  + Automatically track all inventory quantity changes (additions and deductions) over time.
  + Allow Admins to set a "low stock threshold" for each item.
  + Generate a "Reorder Suggestions" list based on two triggers:
    1. Current quantity falls below the user-defined LowStockThreshold.
    2. A rule-based forecast predicts the item will run out soon (e.g., within 7 days).
  + The suggestion should include a recommended reorder quantity based on historical usage.
  + Allow users to manually override or dismiss suggestions.
* **Detailed Implementation Guide:**
  + **Database Entity:**
    1. UsageHistory: Id (UUID, Primary Key), ItemId (UUID, Foreign Key), UserId (UUID, Foreign Key), ChangeDate (DateTime, NOT NULL), QuantityChange (Integer, NOT NULL), Reason (Text, e.g., "Manual Update", "Delivery", "Daily Use").
  + **Logic Implementation:**
    1. The SuggestionService in the Core library will contain the reordering logic.
    2. **Trigger 1 (Threshold):** A background task on app startup will query all items where Quantity <= LowStockThreshold.
    3. **Trigger 2 (Forecasting):** The service will calculate the average daily consumption for each item over the last 30 days from the UsageHistory table.
       - Pseudocode: AvgDailyUse = SUM(QuantityChange WHERE Change < 0) / 30
       - Pseudocode: DaysRemaining = CurrentQuantity / AvgDailyUse
       - If DaysRemaining is less than a configurable value (e.g., 7 days), a reorder suggestion is triggered.
    4. **Recommended Quantity:** The suggestion should recommend reordering enough to last for a set period (e.g., 30 days).
       - Pseudocode: RecommendedOrder = AvgDailyUse \* 30
  + **User Interaction:** If a user deletes an item that is on the reorder list, the system should automatically and silently remove the corresponding suggestion from the list to prevent confusion. The UsageHistory for the deleted item is preserved for historical reporting.

**Post-MVP Feature: Barcode Scanner Integration**

* **Feature Goal:** Accelerate the process of finding and updating inventory items by integrating support for standard USB barcode scanners.
* **API Relationships:** This interacts with the OS-level Human Interface Device (HID) events and the application's inventory search and update functions.
* **Detailed Feature Requirements:**
  + Associate a unique barcode value with each inventory item.
  + When in the main inventory view, scanning a barcode should instantly filter the list to that single item.
  + Provide a dedicated "Scan Mode" for rapid quantity updates (e.g., scan item, enter quantity, repeat).
* **Detailed Implementation Guide:**
  + **Database Schema Change:** Add a Barcode (Text, UNIQUE) field to the Items table.
  + **Implementation:** Most USB barcode scanners emulate a keyboard. The application will listen for rapid keyboard input that ends with an Enter keypress.
  + A global keyboard listener can be configured to capture this input from anywhere in the app. When a valid barcode-like input is detected, it will trigger a search or select action for the item with the matching Barcode value.
  + The "Scan Mode" UI would be a simple view with a text field that has focus by default. After a scan, the app would look up the item and prompt for the quantity to add or subtract.

**System Diagram (MVP)**



**Architecture Consideration Answers**

* **Data Migration for Cloud Sync:**
  + **Solution:** We will use a robust database migration tool from the start. **Entity Framework Core Migrations** is the perfect choice as it's built into the recommended stack.
  + **Process:**
    1. Every change to the database schema (e.g., adding the Barcode column) is scripted into a new migration file.
    2. The application, on startup, checks the database file's migration version against the code's version. If the database is outdated, it automatically applies the pending migration scripts in order.
    3. When we introduce cloud sync, we will add new tables (e.g., SyncLog) via a new migration. The schema's use of **UUIDs** and **timestamps** (CreatedAt/UpdatedAt) is the most critical prerequisite, and it's already designed in. This design ensures that data from multiple local clients can be merged into a central database without key collisions.
* **Data Conflict Prevention (Locking):**
  + **Solution:** For a local-first application, concurrent access by multiple users on the *same machine* is not a typical scenario. The primary risk is a background process conflicting with a user action. We will use **Optimistic Concurrency Control**.
  + **Process:**
    1. Add a Version or RowVersion (timestamp or sequential number) field to the Items table.
    2. When a user opens an item to edit, the application reads and stores the current Version number.
    3. When the user saves their changes, the UPDATE statement will include a WHERE clause to check if the Version number in the database is still the same as the one that was read.
       - Pseudocode: UPDATE Items SET ... WHERE Id = @Id AND Version = @OriginalVersion
    4. If the number of affected rows is 0, it means another process changed the item in the meantime. The application will then notify the user ("This item was modified by another process. Please refresh and try again.") and reload the data. This is much safer and more performant than pessimistic locking (locking the row).
* **Cross-Device Data Syncing:**
  + **Solution:** While the MVP is purely local, we can architect for a future "sync" feature by choosing a suitable, free, and efficient backend. A lightweight **self-hosted Web API** built with **ASP.NET Core** is the most viable solution.
  + **Process:**
    1. The desktop app would remain local-first, operating on its SQLite database for speed and offline capability.
    2. A new "Sync" button or periodic background service in the app would call the self-hosted API.
    3. The sync logic would be:
       - **Push:** Send all local records where UpdatedAt is newer than the LastSyncTimestamp to the server.
       - **Pull:** Ask the server for all records that have been updated since the LastSyncTimestamp.
       - The server API would handle merging data into a central database (e.g., PostgreSQL, another SQLite file). The use of UUIDs makes this process robust. This hybrid approach provides the best of both worlds: the responsiveness of a local app and the data ubiquity of the cloud, without forcing a constant internet connection.